Chapter 6. The Shell

Introduction

In this chapter we will look at one of the major component of UNIX architecture – The Shell. Shell acts as both a command interpreter as well as a programming facility. We will look at the interpretive nature of the shell in this chapter.

Objectives

- The Shell and its interpretive cycle
- Pattern Matching – The wild-cards
- Escaping and Quoting
- Redirection – The three standard files
- Filters – Using both standard input and standard output
- /dev/null and /dev/tty – The two special files
- Pipes
- tee – Creating a tee
- Command Substitution
- Shell Variables

1. The shell and its interpretive cycle

The shell sits between you and the operating system, acting as a command interpreter. It reads your terminal input and translates the commands into actions taken by the system. The shell is analogous to ***command.com*** in DOS. When you log into the system you are given a default shell. When the shell starts up it reads its startup files and may set environment variables, command search paths, and command aliases, and executes any commands specified in these files. The original shell was the Bourne shell, ***sh***. Every Unix platform will either have the Bourne shell, or a Bourne compatible shell available.

Numerous other shells are available. Some of the more well known of these may be on your Unix system: the Korn shell, ***ksh***, by David Korn, C shell, ***csh***, by Bill Joy and the Bourne Again SHell, ***bash***, from the Free Software Foundations GNU project, both based on ***sh***, the T-C shell, ***tcsh***, and the extended C shell, ***cshe***, both based on ***csh***.

Even though the shell appears not to be doing anything meaningful when there is no activity at the terminal, it swings into action the moment you key in something.

The following activities are typically performed by the shell in its interpretive cycle:

- The shell issues the prompt and waits for you to enter a command.
- After a command is entered, the shell scans the command line for metacharacters and expands abbreviations (like the * in rm *) to recreate a simplified command line.
- It then passes on the command line to the kernel for execution.

- The shell waits for the command to complete and normally can't do any work while the command is running.
- After the command execution is complete, the prompt reappears and the shell returns to its waiting role to start the next cycle. You are free to enter another command.

2. Pattern Matching – The Wild-Cards

A pattern is framed using ordinary characters and a metacharacter (like *) using well-defined rules. The pattern can then be used as an argument to the command, and the shell will expand it suitably before the command is executed.

The metacharacters that are used to construct the generalized pattern for matching filenames belong to a category called wild-cards. The following table lists them:

Wild-Card	Matches
*	Any number of characters including none
?	A single character
[ijk]	A single character – either an i, j or k
[x-z]	A single character that is within the ASCII range of characters x and x
[!ijk]	A single character that is not an i,j or k (Not in C shell)
[!x-z]	A single character that is not within the ASCII range of the characters x and x (Not in C Shell)
{pat1,pat2…}	Pat1, pat2, etc. (Not in Bourne shell)

Examples:

To list all files that begin with *chap*, use

```
$ ls chap*
```

To list all files whose filenames are six character long and start with chap, use

```
$ ls chap??
```

Note: Both * and ? operate with some restrictions. for example, the * doesn't match all files beginning with a . (dot) ot the / of a pathname. If you wish to list all hidden filenames in your directory having at least three characters after the dot, the dot must be matched explicitly.

```
$ ls .???*
```

However, if the filename contains a dot anywhere but at the beginning, it need not be matched explicitly.

Similarly, these characters don't match the / in a pathname. So, you cannot use

```
$ cd /usr?local          to change to /usr/local.
```

The character class

You can frame more restrictive patterns with the character class. The character class comprises a set of characters enclosed by the rectangular brackets, [and], but it matches a single character in the class. The pattern [abd] is character class, and it matches a single character – an a,b or d.

Examples:

```
$ls chap0[124]          Matches chap01, chap02, chap04 and lists if found.
```

$ ls chap[x-z] Matches chapx, chapy, chapz and lists if found.

You can negate a character class to reverse a matching criteria. For example,

- To match all filenames with a single-character extension but not the .c ot .o files, use *.[!co]
- To match all filenames that don't begin with an alphabetic character, use [!a-zA-Z]*

Matching totally dissimilar patterns

This feature is not available in the Bourne shell. To copy all the C and Java source programs from another directory, we can delimit the patterns with a comma and then put curly braces around them.

$ cp $HOME/prog_sources/*.{c,java} .

The Bourne shell requires two separate invocations of cp to do this job.

$ cp /home/srm/{project,html,scripts/* .

The above command copies all files from three directories (project, html and scripts) to the current directory.

3. Escaping and Quoting

Escaping is providing a \ (backslash) before the wild-card to remove (escape) its special meaning.

For instance, if we have a file whose filename is chap* (Remember a file in UNIX can be names with virtually any character except the / and null), to remove the file, it is dangerous to give command as *rm chap**, as it will remove all files beginning with chap. Hence to suppress the special meaning of *, use the command *rm chap**

To list the contents of the file chap0[1-3], use

$ cat chap0\[1-3\]

A filename can contain a whitespace character also. Hence to remove a file named My Documend.doc, which has a space embedded, a similar reasoning should be followed:

$ rm My\ Document.doc

Quoting is enclosing the wild-card, or even the entire pattern, within quotes. Anything within these quotes (barring a few exceptions) are left alone by the shell and not interpreted.

When a command argument is enclosed in quotes, the meanings of all enclosed special characters are turned off.

Examples:

$ rm 'chap*' Removes fil *chap**

$ rm "My Document.doc" Removes file *My Document.doc*

4. Redirection : The three standard files

The shell associates three files with the terminal – two for display and one for the keyboard. These files are streams of characters which many commands see as input and output. When a user logs in, the shell makes available three files representing three streams. Each stream is associated with a default device:

Standard input: The file (stream) representing input, connected to the keyboard.

Standard output: The file (stream) representing output, connected to the display.
Standard error: The file (stream) representing error messages that emanate from the command or shell, connected to the display.

The standard input can represent three input sources:
The keyboard, the default source.
A file using redirection with the < symbol.
Another program using a pipeline.

The standard output can represent three possible destinations:
The terminal, the default destination.
A file using the redirection symbols > and >>.
As input to another program using a pipeline.

A file is opened by referring to its pathname, but subsequent read and write operations identify the file by a unique number called a file descriptor. The kernel maintains a table of file descriptors for every process running in the system. The first three slots are generally allocated to the three standard streams as,

0 – Standard input
1 – Standard output
2 – Standard error

These descriptors are implicitly prefixed to the redirection symbols.
Examples:
Assuming file2 doesn't exist, the following command redirects the standard output to file *myOutput* and the standard error to file *myError*.

$ ls –l file1 file2 1>myOutput 2>myError

To redirect both standard output and standard error to a single file use:

$ ls –l file1 file2 1>| myOutput 2>| myError OR
$ ls –l file1 file2 1> myOutput 2>& 1

5. Filters: Using both standard input and standard output

UNIX commands can be grouped into four categories viz.,

1. Directory-oriented commands like mkdir, rmdir and cd, and basic file handling commands like cp, mv and rm use neither standard input nor standard output.
2. Commands like ls, pwd, who etc. don't read standard input but they write to standard output.
3. Commands like lp that read standard input but don't write to standard output.
4. Commands like cat, wc, cmp etc. that use both standard input and standard output.

Commands in the fourth category are called filters. Note that filters can also read directly from files whose names are provided as arguments.
Example: To perform arithmetic calculations that are specified as expressions in input file calc.txt and redirect the output to a file result.txt, use

$ bc < calc.txt   > result.txt

6. /dev/null and /dev/tty : Two special files

/dev/null: If you would like to execute a command but don't like to see its contents on the screen, you may wish to redirect the output to a file called /dev/null. It is a special file that can accept any stream without growing in size. It's size is always zero.

/dev/tty: This file indicates one's terminal. In a shell script, if you wish to redirect the output of some select statements explicitly to the terminal. In such cases you can redirect these explicitly to /dev/tty inside the script.

7. Pipes

With piping, the output of a command can be used as input (piped) to a subsequent command.

```
$ command1 | command2
```

Output from command1 is piped into input for command2.

This is equivalent to, but more efficient than:

```
$ command1 > temp
$ command2 < temp
$ rm temp
```

Examples

```
$ ls -al | more
$ who | sort | lpr
```

When a command needs to be ignorant of its source

If we wish to find total size of all C programs contained in the working directory, we can use the command,

```
$ wc –c *.c
```

However, it also shows the usage for each file(size of each file). We are not interested in individual statistics, but a single figure representing the total size. To be able to do that, we must make wc ignorant of its input source. We can do that by feeding the concatenated output stream of all the .c files to wc –c as its input:

```
$ cat *.c | wc –c
```

8. Creating a tee

tee is an external command that handles a character stream by duplicating its input. It saves one copy in a file and writes the other to standard output. It is also a filter and hence can be placed anywhere in a pipeline.

Example: The following command sequence uses tee to display the output of who and saves this output in a file as well.

```
$ who | tee users.lst
```

9. Command substitution

The shell enables the connecting of two commands in yet another way. While a pipe enables a command to obtain its standard input from the standard output of another command, the shell enables one or more command arguments to be obtained from the standard output of another command. This feature is called command substitution.

Example:

```
$ echo Current date and time is `date`
```

Observe the use of backquotes around date in the above command. Here the output of the command execution of date is taken as argument of echo. The shell executes the enclosed command and replaces the enclosed command line with the output of the command.

Similarly the following command displays the total number of files in the working directory.

```
$ echo "There are 'ls | wc –l' files in the current directory"
```

Observe the use of double quotes around the argument of echo. If you use single quotes, the backquote is not interpreted by the shell if enclosed in single quotes.

10. Shell variables

Environmental variables are used to provide information to the programs you use. You can have both global environment and local shell variables. Global environment variables are set by your login shell and new programs and shells inherit the environment of their parent shell. Local shell variables are used only by that shell and are not passed on to other processes. A child process cannot pass a variable back to its parent process.

To declare a local shell variable we use the form *variable=value* (no spaces around =) and its evaluation requires the $ as a prefix to the variable.

Example:

```
$  count=5
$  echo $count
5
```

A variable can be removed with **unset** and protected from reassignment by **readonly**. Both are shell internal commands.

Note: In C shell, we use **set** statement to set variables. Here, there either has to be whitespace on both sides of the = or none at all.

```
$ set count=5
$ set size = 10
```

Uses of local shell variables

1. Setting pathnames: If a pathname is used several times in a script, we can assign it to a variable and use it as an argument to any command.
2. Using command substitution: We can assign the result of execution of a command to a variable. The command to be executed must be enclosed in backquotes.
3. Concatenating variables and strings: Two variables can be concatenated to form a new variable.

Example: $ base=foo ; ext=.c
$ file=$base$ext
$ echo $file // prints foo.c

Conclusion

In this chapter we saw the major interpretive features of the shell. The following is a summary of activities that the shell performs when a command line is encountered at the prompt.

- Parsing: The shell first breaks up the command line into words using spaces and tabs as delimiters, unless quoted. All consecutive occurrences of a space or tab are replaced with a single space.
- Variable evaluation: All $-prefixed strings are evaluated as variables, unless quoted or escaped.
- Command substitution: Any command surrounded by backquotes is executed by the shell, which then replaces the standard output of the command into the command line.
- Redirection: The shell then looks for the characters >, < and >> to open the files they point to.
- Wild-card interpretation: The shell then scans the command line for wild-cards (the characters *, ?, [and]). Any word containing a wild-card is replaced by a sorted list of filenames that match the pattern. The list of these filenames then forms the arguments to the command.
- PATH evaluation: It finally looks for the PATH variable to determine the sequence of directories it has to search in order to find the associated binary.